Untersuchung von Deep Q-Learning und Soft Actor-Critic auf der BipedalWalker-v3 Umgebung

1st Nandino Cakar
Hochschule Bochum
Heiligenhaus, Deutschland
nandino.cakar@stud.hs-bochum.de

2nd Jan Barthel
Hochschule Bochum
Heiligenhaus, Deutschland
jan.barthel@stud.hs-bochum.de

***Abstract*—In diesem Paper haben wir zwei Algorithmen des bestärkenden Lernens implementiert: Deep Q-Learning und Soft Actor-Critic. Das Ziel ist es mithilfe der beiden Verfahren die BipedalWalker-v3 Umgebung des OpenAI-Gyms zu lösen. Der von uns implementierte Deep Q-Learning Algorithmus stagniert bei einer negativen Belohnung und schafft es nicht dem Agenten das Laufen beizubringen. Mithilfe des Soft Actor-Critic-Verfahrens konnte der Agent so trainiert werden, dass er die Umgebung innerhalb von 474 Episoden löst. Des Weiteren haben wir einen Soft-Actor-Critic Agenten in der BipedalWalkerHardcore-v3 Umgebung implementiert, der es regelmäßig schafft Rewards von über 300 zu erreichen.**

I. Einleitung

Das Ziel dieser Arbeit ist es, die BipedalWalker-v3 Umgebung von OpenAI-Gym mithilfe von zwei unterschiedlichen Verfahren des bestärkenden Lernens zu lösen. Bei dem ersten Verfahren handelt es sich um das Deep Q-Learning, welches uns aus der Vorlesung bekannt ist. Als zweites Verfahren wird der Soft Actor-Critic Algorithmus angewendet, welcher zu den modernsten Lernverfahren gehört und verschiedene Techniken des bestärkenden Lernens kombiniert. In dieser Arbeit werden beide Verfahren implementiert und deren Ergebnisse auf der BipedalWalker-v3 Umgebung ausgewertet.

Hierzu wird zunächst in Kapitel II die Umgebung des BipedalWalkers-v3 und des BipedalWalkersHardcore-v3 aus dem OpenAI-Gym beschrieben. In Kapitel III folgen daraufhin die Grundlagen der zwei bestärkenden Lernverfahren. Zu den Grundlagen des Deep Q-Learning zählen das Q-Learning, das Q-Learning mit neuronalen Netzen sowie das Double Q-Learning. Anschließend werden die Policy Optimization und die Actor-Critic-Architektur beschrieben. Diese bilden wiederum die Grundlage für das eingesetzte Soft Actor-Critic Verfahren. Nachfolgend geht es in Kapitel IV um die Implementierung der beiden Verfahren. Hierzu wird die Umsetzung der Algorithmen im Detail erklärt. Die Ergebnisse werden in Kapitel V dargestellt und diskutiert. Abschließend werden die Erkenntnisse zusammengefasst und Möglichkeiten der Optimierung vorgestellt.

II. Environment

In diesem Abschnitt soll die BipedalWalker-v3 Umgebung genauer erläutert werden. Der BipedalWalker ist eine von vielen Testumgebungen im OpenAI-Gym, welche zum Testen verschiedener Algorithmen genutzt werden kann. Bei dem BipedalWalker handelt es sich um einen zweibeinigen Roboter, der versucht das Laufen zu erlernen. Die Umgebung kann als Framework in Python über das OpenAI-Gym Modul eingebunden und verwendet werden. Neben dem BipedalWalker-v3 gibt es zusätzlich eine ähnliche Umgebung mit erhöhtem Schwierigkeitsgrad, den BipedalWalkerHardcore-v3. Diese beiden Umgebungen werden nachfolgend genauer erläutert.

A. BipedalWalker-v3

In Abbildung 1 ist der Walker in der Umgebung BipedalWalker-v3 dargestellt. Der BipedalWalker besteht aus einem Rumpf mit zwei Beinen, mit denen er Kontakt zum Boden hat (rot). Jedes Bein besitzt zwei Gelenke, eine Kniegelenk (blau) und ein Hüftgelenk (orange).

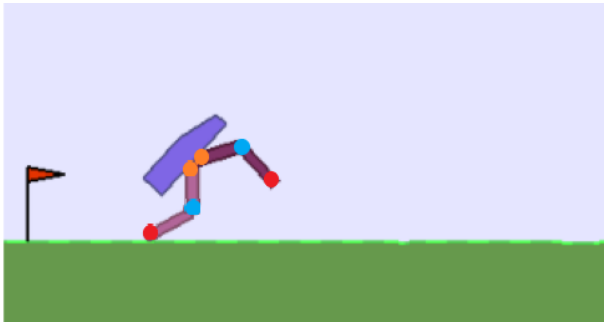


Fig. 1: Der Agent in der BipedalWalker-v3 Umgebung.

Zusätzlich verfügt er über ein Lidar-System mit 10 Lasern zur Abstandsmessung. In Table I ist der gesamte Beobachtungsraum der BipedalWalker-v3 Umgebung abgebildet.

Der Boden, auf dem sich der Walker bewegt, ist eine zufällig generierte Oberfläche, welche Unebenheiten aufweist. Damit der Agent das Laufen erlernen kann, wird jeweils ein Drehmoment auf die vier Gelenke ausgeübt. Der Aktionsraum des Walkers besteht somit aus einem Vektor mit vier Elementen. In jedem Zustand s wird eine bestimmte Aktion a ausgeführt, die in dem Bereich $\mathcal{A} \in [-1, 1]$ liegt. Der Aktionsraum des Walker ist in Table II abgebildet.

Nach jeder Aktion erhält der Agent Feedback durch die Umwelt. Dazu zählen der neue Zustand als auch einen Reward. Die durchgeführte Aktion in dem Zustand s bringt den Agenten in einen neuen Zustand s'. Je nachdem wie gut die

TABLE I: Beobachtungsraum der BipedalWalker-v3 Umgebung [1]

Num	Observation	Min	Max	Mean
0	hull_angle	0	2*pi	0.5
1	hull_angularVelocity	-inf	+inf	-
2	vel_x	-1	+1	-
3	vel_y	-1	+1	-
4	hip_joint_1_angle	-inf	+inf	-
5	hip_joint_1_speed	-inf	+inf	-
6	knee_joint_1_angle	-inf	+inf	-
7	knee_joint_1_speed	-inf	+inf	-
8	leg_1_ground_contact_flag	0	1	-
9	hip_joint_2_angle	-inf	+inf	-
10	hip_joint_2_speed	-inf	+inf	-
11	knee_joint_2_angle	-inf	+inf	-
12	knee_joint_2_speed	-inf	+inf	-
13	leg_2_ground_contact_flag	0	1	-
14-23	10 lidar readings	-inf	+inf	-

TABLE II: Aktionsraum der BipedalWalker-v3 Umgebung [1]

Num	Name	Min	Max
0	Hip_1 (Torque / Velocity)	-1	1
1	Knee_1 (Torque / Velocity)	-1	1
2	Hip_2 (Torque / Velocity)	-1	1
3	Knee_2 (Torque / Velocity)	-1	1

gewählte Aktion in dem Zustand s ist, gibt die Umwelt einen entsprechenden hohen oder niedrigen Reward zurück. Das Ziel ist es, eine Policy π zu lernen, durch die der Reward maximiert wird.

Der Walker startet am linken Ende der Umgebung mit einem geraden Rumpf und mit beiden Beinen in der gleichen Position. Für jede Bewegung der Gelenke erhält der Agent einen kleinen negativen Reward, der proportional zu dem eingesetzten Drehmoment τ ist. Dies soll dazu führen, eine möglichst effiziente Bewegung zu lernen. Schafft es der Walker sich vorwärts zu bewegen, erhält er proportional zur zurückgelegten Distanz Δx einen positiven Reward. Um den Rumpf des Walkers zu stabilisieren bzw. die Geradlinigkeit zu behalten, erhält er ebenfalls eine kleine negative Belohnung, die proportional zum Rumpfwinkel $hull\_angle$ ist. Fällt der Walker hin, erhält er einen Reward von -100. Läuft der Walker effizient bis zum Ende der Umgebung, kann er einen Reward von über 300 erreichen. Erreicht der Walker einen durchschnittlichen Reward über 100 Episoden von mindestens 300, gilt die Umgebung als gelöst.

Zusammenfassend lässt sich die Reward-Funktion wie folgt beschreiben [2]:

$$r = \begin{cases} -100, & \text{wenn fällt} \\ 130 \cdot \Delta x - 5 \cdot |hull\_angle| - 0.00035 \cdot |\tau|, & \text{sonst} \end{cases} \quad (1)$$

B. BipedalWalkerHardcore-v3

Bei der BipedalWalkerHardcore-v3 Umgebung handelt es sich eine Erweiterung des BipedalWalker-v3 mit einem erhöhten Schwierigkeitsgrad. Zusätzliche Hindernisse in der Form von Treppenstufen, Gruben und Stolperfallen sorgen dafür, dass diese Umgebung deutlich anspruchsvoller ist. Ein Ausschnitt der BipedalWalkerHardcore-v3 Umgebung ist in Abbildung 2 dargestellt.

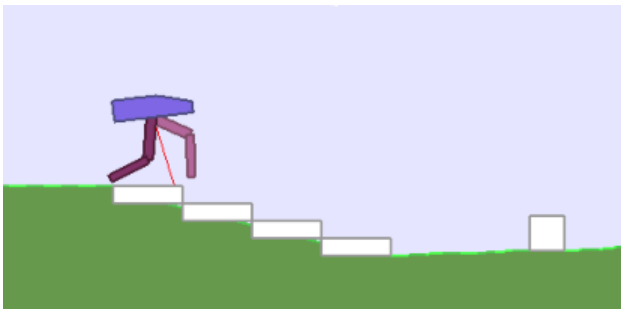


Fig. 2: Der Agent in der BipedalWalkerHardcore-v3 Umgebung.

Während in der BipedalWalker-v3 Umgebung das Lidar-System eher eine untergeordnete Rolle spielt, ist es hier entscheidend, um die Hindernisse identifizieren zu können. Dies hilft dem Agenten bei der Auswahl der richtigen Aktionen, um diese Hindernisse überwinden und so ein Hinfallen zu verhindern.

III. Grundlagen

In diesem Abschnitt werden die Grundlagen zum Deep Q-Learning und dem Soft-Actor-Critic Algorithmus beschrieben. Hierzu werden zu Beginn die Grundlagen des Q-Learnings und der Policy-Optimization aufgefasst. Diese bilden wiederum die Grundlage für das Soft-Actor-Critic Verfahren.

A. Q-Learning

Die Grundlagen des Q-Learnings orientieren sich inhaltlich an den Kapiteln 14 und 15 aus dem Buch Maschinelles Lernen von Prof. Dr. rer. nat. Jörg Frochte [3]. Im Bereich des maschinellen Lernens gibt es unterschiedliche Lernverfahren. Das *Q-Learning* gehört zu den Methoden des bestärkenden Lernens (engl. Reinforcment Learning). Dabei ist das Eingabemuster bekannt, die Zielwerte unbekannt und der Agent lernt nach dem Trial-and-Error Prozess. Je nachdem, ob die Aktion in dem aktuellen Zustand gut oder schlecht ist, erhält er einen guten oder schlechten Reward. Durch den Reward soll der Agent die optimale Policy unter Verwendung der Q-Funktion finden. Das Ziel ist also eine Policy π zu erlernen, die den maximalen kumulierten Reward erreicht. Die optimale Policy π^* ist abhängig von der Value-Funktion V^π, die wie folgt definiert ist:

$$V^\pi(s_t) = \sum_{i=0}^{\infty} \gamma^i \; r_{t+i}, \quad (2)$$

wobei γ der Diskontierungsfaktor ist, der die Gewichtung des Rewards bestimmt. Der Diskontierungsfaktor ist ein konstanter Parameter im Bereich von $\gamma \in [0, 1]$. Liegt der Wert nahe 1 wertet der Agent zukünftige Belohnungen mit größerem Gewicht und ist er nahe 0 werden unmittelbare Belohnungen höher gewichtet. Mithilfe der Value-Funktion ergibt sich für die optimale Policy π^*:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \; f\ddot{u}r \; alle \; s \in S \tag{3}$$

Q-Learning basiert darauf, eine Action-Value-Funktion $Q(s, a)$ zu lernen, um eine optimale Policy für die Auswahl von Aktionen zu lernen. Die Action-Value-Funktion liefert den erwarteten Nutzen einer Aktion a in einem bestimmten Zustand s. Die Action-Value-Funktion $Q(s, a)$ ist wie folgt definiert [4]:

$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a)) \tag{4}$$

Um die optimale Policy zu finden, werden die Aktionen in den Zuständen jeweils variiert. Nach dem Optimalitätsprinzip von Bellman setzt sich die optimale Policy aus verschiedenen Sequenzen optimaler Teillösungen zusammen. Somit wird die Aktion ausgewählt, die den Q-Wert für den nächsten Zeitschritt maximiert. Die Gleichung wird wie folgt definiert:

$$Q_{neu} = (1 - \alpha)Q_{alt}(s, a) + \alpha \cdot (r + \gamma \cdot \underset{a'}{max} Q(s', a')), \tag{5}$$

wobei α die Lernrate, s der gegenwärtige Zustand, s' der nächste Zustand, a die gegenwärtige Aktion und a' die nächste Aktion ist.

1) Q-Learning mit neuronalen Netzen: Das Q-Learning lässt sich auch mithilfe von neuronalen Netzen umsetzen. Dies bietet sich vor allem bei großen bzw. kontinuierlichen Zustandsräumen an. Bisher werden die Erfahrungen des Agenten bestehend aus (s, a, s', r, d) nicht gespeichert und somit nach jedem Schritt in der Umgebung verworfen. Anstatt diese Informationen zu verwerfen, kann es von Vorteil sein diese in einem Pufferspeicher abzulegen. Dieser Speicher wird auch *Replay Buffer* bezeichnet. Die Erfahrungen aus dem Replay Buffer können anschließend genutzt werden, um den Agenten wiederholt mit diesen Daten zu trainieren. Diese Technik wird auch als *Experience Replay* bezeichnet und ermöglicht eine schnellere Konvergenz des neuronalen Netzes [5]. Anstatt jedes mal mit dem ganzen Replay Buffer zu lernen, werden i. d. R. sogenannte *Mini-Batches* eingesetzt.

Durch das eingesetzte neuronale Netz und dem Mini-Batch entstehen jedoch auch Probleme. Dem Netz werden zu Beginn eine bestimmte Menge an Freiheitsgraden zugewiesen und der Mini-Batch erhält ebenfalls eine bestimmte Größe. Besitzt das Netz zu wenig Freiheitsgerade und der Mini-Batch ist zu klein gewählt, werden ungewollt Werte an anderen Stellen des neuronalen Netzes verändert. Werden die Freiheitsgerade und der Mini-Batch jedoch zu groß gewählt, kann es zum Overfitting des Netzes führen. Somit generalisiert der Agent nicht mehr und lernt nur noch auswendig.

2) Double Q-Learning: Das Q-Learning mit einem neuronalen Netz neigt dazu, durch eine Verwendung der gleichen Datengrundlage, den Wert von Zustands-Aktions-Paaren zu überschätzen. Denn die Datengrundlage wird einerseits verwendet, um die Aktion zu wählen, die die höchste Belohnung bringt und andererseits um den Aktion-Value-Wert zu approximieren. Um das Überschätzen des Q-Wertes zu vermeiden, entstand die Idee des Double Q-Learnings von Hado van Hasselt [6]. Beim Double Q-Learning werden zwei unterschiedliche Q-Funktionen verwendet. Es gibt zum einen ein Netz Q und ein Target-Netz Q' [7]. Die Q-Funktion Q wird zum auswählen der Aktionen genutzt und die zweite Funktion Q' für die Aktionsbewertung. In bestimmten Intervallen werden die Parameter von dem Netz Q auf das Target-Netz Q' kopiert. Dieser Prozess macht das Q-Learning robuster, da nun beide Netze an derselben Stelle den Wert überschätzen müssten. Ein Nebeneffekt ist, dass das Training deutlich länger dauert, da zwei neuronale Netze trainiert werden müssen. Es ist zu beachten, dass sich die beiden Netze nicht zu ähnlich werden, weil sie sich sonst nicht mehr gegenseitig kontrollieren können.

3) Deep Q-Learning: DeepMind hat im Jahr 2015 zum ersten mal das Deep Q-Learning in einem Paper vorgestellt [8]. Hierbei ist die Neuerung nicht das Benutzen von tiefen Netzen, denn diese konnten bereits bei den vorherigen Verfahren eingesetzt werden. Beim Deep Q-Learning geht es um die Zusammensetzung der Q-Funktion, welche sich im Vergleich zu den vorherigen Verfahren ändert. Während bisher immer ein Vektor aus dem aktuellen Zustand s und der Aktion a als Input übergeben wird, ändert sich dies beim Deep Q-Learning. Anstatt einem Vektor aus Zustand und Aktion, erhält der Agent nur noch den aktuellen Zustand und gibt für jede Aktion in diesem Zustand die erwartete Belohnung zurück (siehe Abbildung 3).

Zusätzlich wird zu dieser neuen Technik das zuvor beschrieben Experience Replay und ein Target-Network zur Stabilisierung eingesetzt. Um gute Ergebnisse zu erreichen sind viele Iterrationsschritte nötig. Daher bietet es sich an mit Mini-Batches zu arbeiten, wodurch die Trainingszeit verkürzt werden kann. Dabei besteht allerdings die Gefahr, dass es zu einem Overfitting genau auf die Bestandteile des Mini-Batches kommt. Um dem entgegenzuwirken ist es sinnvoll eine kleine Lernrate zu wählen. Der Algorithmus aus dem Paper von DeepMind ist in Algorithm 1 dargestellt. Es ist dabei zu beachten, dass in der vorgestellten Version des Deep Q-Learnings mit Bildern gelernt wurde, wodurch der Algorithmus nicht vollständig auf die BipedalWalker Umgebung übertragen werden kann.

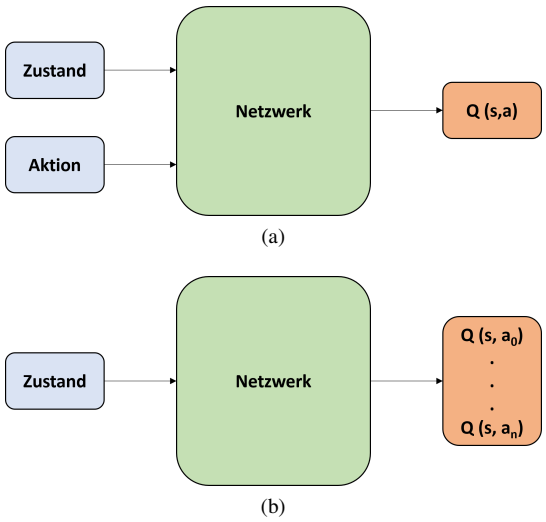


Fig. 3: Netzarchitektur des klassischen Q-Learnings (a) und des Deep Q-Learnings (b) (angelehnt an [3])

Algorithm 1 Deep Q-Learning

1: Initialisiere ein leeres Replay Buffer D
2: Initialisiere die Action-Value-Funktion Q mit zufälligen Gewichten θ
3: Initialisiere die Target Action-Value-Funktion $\hat{Q}$ mit der Kopie θ' von θ
4: **for** episode = 1...M **do**
5: Setze die Umgebung zurück
6: Initialisiere eine Sequenz $s_1 = \{x_1\}$ und Preprocessing-Sequenz $\phi_1 = \phi(s_1)$
7: **for** t = 1...T **do**
8: Wähle mit einer Wahrscheinlichkeit von ϵ eine zufällige Aktion, sonst wähle

$$a_t = \underset{a}{argmax} Q_\theta(\phi(s_t), a) \tag{6}$$

9: Der Agent führt die Aktion a_t aus
10: Der Agent beobachtet o_{t+1} und erhält einen Reward r_t
11: Setze $s_{t+1} = \{s_t, a_t, o_{t+1}\}$ und führe Preprocessing $\phi_{t+1} = \phi_(s_{t+1})$ durch
12: Speichere $(\phi_t, a_t, r_t, \phi_{t+1})$ im Replay Buffer D
13: Wähle einen zufälligen Mini-Batch $(\phi_j, a_j, r_j, \phi_{j+1})$ aus D
14: $y_j = \begin{cases} r_j, & \text{wenn } done_{j+1} \\ r_j + \gamma \underset{a'}{max} \hat{Q}'_\theta(\phi_{j+1}, a') & \text{sonst} \end{cases}$
15: Führe einen Optimierungsschritt bzgl. des Fehlers $(y_j - Q_\theta(\phi_j, a_j))^2$ durch
16: Alle C Schritte setze $\hat{Q} = Q$

B. Policy Optimization

Beim Q-Learning-Ansatz wird die optimale Policy indirekt gelernt, indem Action-Value-Werte berechnet und dann die beste Aktion ausgewählt wird. Es wird also eine Policy π gelernt, die in jedem Zustand s die Aktion a mit dem größten Action-Value-Wert besitzt. Es ist allerdings auch möglich die Policy direkt zu optimieren, ohne den Umweg über die Q-Funktion zu gehen. Dies ist zum Beispiel dann sinnvoll, wenn es sich um eine stochastische Umgebung oder eine Umgebung mit einem großen Aktionsraum handelt.

Eine Möglichkeit die Policy direkt zu optimieren ist durch sogenannte *Policy-Gradient-Verfahren*. Policy-Gradient-Verfahren lernen eine parametrisierte Policy π_θ, wobei θ z. B. die Gewichte eines neuronalen Netzes sein können. Das Ziel ist es den erwarten kumulierten Reward durch das Optimieren des Parameters θ zu maximieren. Dazu wird eine neue Größe, die *Policy Performance* $J(\pi_\theta)$ eingeführt:

$$J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathbb{E}}[r(\tau)], \tag{7}$$

wobei τ eine Sequenz von Zuständen und Aktionen ist. Die Optimierung der Policy erfolgt mithilfe des Gradientenaufstiegs:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_t}, \tag{8}$$

wobei α die Schrittweite ist. Der Gradient der Policy Performance $\nabla_\theta J(\pi_\theta)$ wird auch *Policy Gradient* (PG) genannt. Der PG definiert die Richtung, in die der Parameter θ geändert werden muss, sodass die Policy einen höheren kumulierten Reward hervorbringt. Um diesen Algorithmus anwenden zu können, muss der PG so ausgedrückt werden, dass er sich numerisch berechnen lässt. Dazu wird der PG umformuliert zu:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathbb{E}}\left[\sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t)\Phi_t\right], \tag{9}$$

wobei für Φ_t sowohl $r(\tau)$ als auch $Q^{\pi_\theta}(s_t, a_t)$ verwendet werden kann. Auch andere Beschreibungen der zu erwartenden Belohnung sind möglich. Für die mathematische Herleitung wird auf [9] verwiesen. Nachfolgend wird $\Phi_t = Q^{\pi_\theta}(s_t, a_t)$ eingesetzt.

Beim PG wird der Gradient der Log-Wahrscheinlichkeit der ausgeführten Aktion verwendet. Es wird also versucht, die Wahrscheinlichkeit von Aktionen mit einer hohen Belohnung zu erhöhen und die Wahrscheinlichkeiten von Aktionen mit einer schlechten Belohnung zu verringern. Allerdings ist der Wertebereich der Belohnungen abhängig von der Umgebung, in der gelernt werden soll. Dies kann zu Problemen führen, da sich die Belohnungen verschiedener Episoden sehr stark unterscheiden können. Sehr große Unterschiede können sich stark auf das Training auswirken, weil eine außergewöhnlich erfolgreiche Belohnung den endgültigen Gradienten sehr stark beeinflussen kann. Der PG weist somit eine hohe Varianz

auf. Eine Möglichkeit dem entgegenzuwirken ist mithilfe einer sogenannten *Baseline* b, welche von den Q-Werten abgezogen wird. [10]

Das Einfügen der Baseline $b(s_t)$ ist möglich, da für Funktionen, die nicht abhängig von der Aktion sind, gilt:

$$\mathop{\mathbb{E}}_{a_t \sim \pi_\theta}\Big[\nabla_\theta log \pi_\theta(a_t|s_t) b(s_t)\Big] = 0 \tag{10}$$

Somit können solche Funktionen addiert oder subtrahiert werden ohne den Erwartungswert zu verändern:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}\Big[\sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t)\Big(Q^{\pi_\theta}(s_t, a_t) - b(s_t)\Big)\Big] \tag{11}$$

Üblicherweise wird als Baseline die Value-Funktion $V^\pi(s_t)$ verwendet (siehe Gleichung 2), welche das Training beschleunigt und stabiler macht. Der entstehende Term wird auch *Advantage-Funktion* genannt und beschreibt wie viel besser oder schlechter die gewählte Aktion a_t, im Vergleich zur aktuellen Policy ist: [11]

$$A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t) \tag{12}$$

C. Actor-Critic-Architektur

Die Actor-Critic-Methode [12] ist eine Kombination aus den Gebieten der Policybasierten und Wertebasierten Verfahren (siehe Abbildung 4). Wie im letzten Abschnitt erwähnt, kann die Value-Funktion als Baseline verwendet werden, um die Varianz des PG zu verringern. Die Baseline wird typischerweise mit einem neuronalen Netz approximiert, wodurch ebenfalls der Advantage-Wert einer Aktion a_t im Zustand s_t angenähert werden kann. Dieser Ansatz wird auch Actor-Critic-Architektur genannt, wobei die Policy π_θ der Actor und die Baseline b der Critic ist [13].

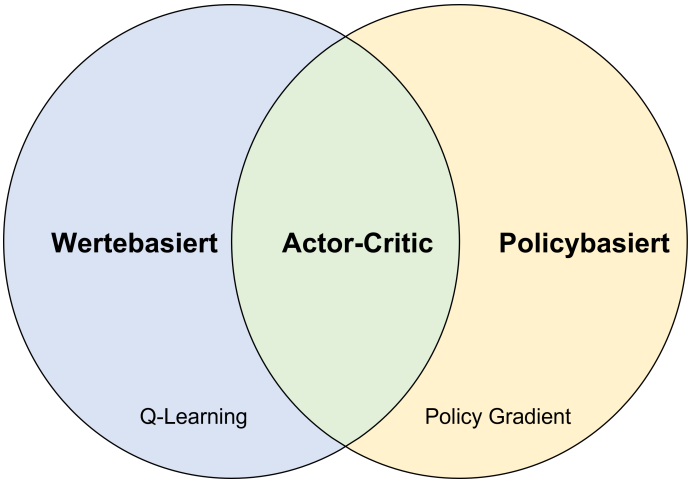


Fig. 4: Einordnung der Actor-Critic-Methode

Die Actor-Critic-Architektur besteht in ihrer einfachsten Form aus zwei neuronalen Netzen: dem Actor-Netz, welches die Policy π_θ lernt und dem Critic-Netz, welches die Action-Value-Funktion Q_ϕ oder die Value-Funktion V_ϕ lernt. Das Actor-Netz liefert für den aktuellen Zustand als Output eine Aktion oder eine Policy [14]. Das Critic-Netz approximiert die Action-Value-Funktion bzw. Value-Funktion auf Basis der aktuellen Policy des Actor-Netzes. Auf diese Weise bewertet der Critic die Policy des Actors. Der Output des Critic-Netzes leitet somit den Lernprozess der beiden Netze an [15]. Abbildung 5 stellt den Aufbau einer Actor-Critic-Architektur grafisch dar.

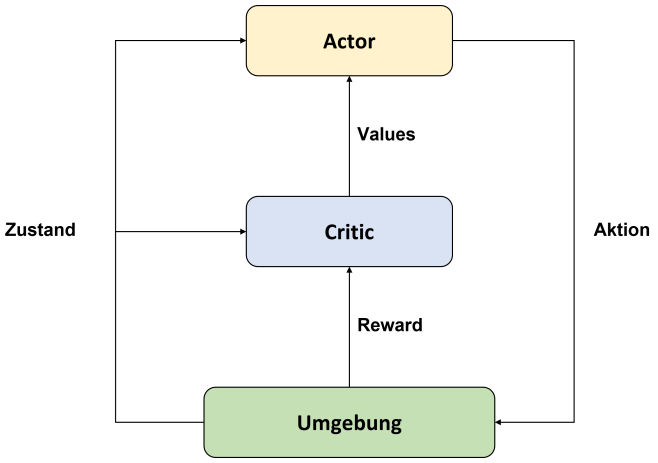


Fig. 5: Actor-Critic-Architektur (angelehnt an [16])

D. Soft Actor-Critic

Beim *Soft Actor-Critic* (SAC) Verfahren , welches 2018 von einer Tuomas Haarnoja et al. veröffentlicht worden ist, handelt es sich um einen off-policy Actor-Critic Algorithmus, der auf der *Entropieregularisierung* basiert [17].

Die Policy wird beim SAC so trainiert, dass sie einen Kompromiss zwischen dem erwarteten Reward und der Entropie maximiert. Die Entropie ist dabei ein Maß dafür, wie Zufällig die Policy ist. Durch die Maximierung der Entropie entstehen mehrere Vorteile: Zum einen wird der Policy ein Anreiz dazu gegeben mehr zu erkunden, während sie eindeutig aussichtslose Wege verwirft. Zum anderen kann die Policy mehrere nahezu optimale Lösungen gleichzeitig erfassen. Kommt es in einer Situation dazu, dass mehrere Aktionen gleich gut erscheinen, so wird die Policy für beide Aktionen die gleiche Wahrscheinlichkeit vorhersagen. [18]

Die Entropie $\mathcal{H}$ einer Variable x lässt sich durch die Verteilung der Wahrscheinlichkeitsfunktion P wie folgt berechnen [19]:

$$\mathcal{H}(P) = \mathop{\mathbb{E}}_{x \sim P}[-log P(x)] \tag{13}$$

Durch die Kompromiss zwischen dem erwarteten Reward und der Entropie verändert sich die optimale Policy π^* zu:

$$\pi^* = \underset{\pi}{argmax} \underset{\tau\sim\pi}{\mathbb{E}}\Big[\sum_{t=0}^{\infty}\gamma^t\Big(r(s_t,a_t,s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s_t))\Big)\Big], \tag{14}$$

wobei $\alpha \geq 0$ der Temperaturparameter ist, welcher die Gewichtung der Entropie gegenüber des Rewards bestimmt [18]. Daraus resultiert auch eine Veränderung der Value-Funktion $V^\pi(s)$ und der Action-Value-Funktion $Q^\pi(s,a)$ über jeden Zeitschritt, da die Belohnung für die Entropie hinzugefügt wird [19]:

$$V^\pi(s) = \underset{\tau\sim\pi}{\mathbb{E}}\Big[\sum_{t=0}^{\infty}\gamma^t\Big(r(s_t,a_t,s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s_t))\Big)\Big] \tag{15}$$

$$Q^\pi(s,a) = \underset{\tau\sim\pi}{\mathbb{E}}\Big[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}H(\pi(\cdot|s_t))\Big] \tag{16}$$

V^π lässt sich dann durch Q^π wie folgt ausdrücken [19]:

$$V^\pi(s) = \underset{a\sim\pi}{\mathbb{E}}[Q^\pi(s,a)] + \alpha H(\pi(\cdot|s)) \tag{17}$$

Die Bellman-Gleichung für Q^π ergibt sich dann zu:

$$Q^\pi(s,a) = \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathbb{E}}\Big[r(s,a,s') + \gamma(Q^\pi(s',a') + \alpha\mathcal{H}(\pi(\cdot|s')))\Big] \tag{18}$$

Mithilfe der Gleichung (13) lässt sich die Bellman-Gleichung umschreiben zu:

$$Q^\pi(s,a) = \underset{\substack{s'\sim P\\ a'\sim\pi}}{\mathbb{E}}\Big[r(s,a,s') + \gamma(Q^\pi(s',a') - \alpha log\pi(a'|s'))\Big], \tag{19}$$

wobei $s' \sim P$ bedeutet, dass der nächste Zustand s' von der Umgebung über die Wahrscheinlichkeitsverteilung $P(\cdot|s,a)$ gesampled wird.

Die rechte Seite der Gleichung entspricht der Erwartung über die nächsten Zustände s', die aus dem Replay Buffer stammen und die nächsten Aktionen a', die aus der aktuellen Policy π stammen. Da es sich um einen Erwartungswert E handelt, kann die Approximation über Samples erfolgen:

$$Q^\pi(s,a) \approx r + \gamma(Q^\pi(s',a') - \alpha log\pi(a'|s')), \tag{20}$$

wobei $a' \sim \pi(\cdot|s')$ aus der Policy gesampled wird und nicht aus dem Replay Buffer. Zur Approximation der Action-Value-Funktion wird wie beim Deep Q-Learning ein neuronales Netz $Q_\phi(s,a)$ mit dem Parameter ϕ verwendet und ein Target-Netz $Q_{\phi'}$ mit dem Parameter ϕ'. Anders als beim Deep Q-Learning werden die Gewichte des Target-Netzes nicht nach einer bestimmten Anzahl von Schritten kopiert, sondern bei jedem Update über das sogenannte *Polyak Averaging* angepasst:

$$\phi' \leftarrow \rho\phi' + (1-\rho)\phi, \tag{21}$$

wobei $\rho \in [0,1]$ ein Hyperparameter ist, der typischerweise nahe 1 gewählt wird. [20]
SAC nutzt hier allerdings zusätzlich das sogenannte *Clipped Double Q-Learning*, welches 2018 von Fujimoto et al. [21] veröffentlicht worden ist. Beim Clipped Double Q-Learning werden zwei neuronale Netze Q_{ϕ_1} und Q_{ϕ_2} zur Approximation der Action-Value-Funktion verwendet. Zusätzlich gibt es noch zwei Target-Netze $Q_{\phi'_1}$ und $Q_{\phi'_2}$. Der Zielwert wird dort wie folgt berechnet:

$$y = r + \gamma \underset{i=1,2}{min} Q_{\phi'_i}(s',a') \tag{22}$$

Durch die Verwendung des Minimums der beiden Target-Q-Funktionen wird immer der kleinere kumulierte Reward ausgewählt. Eine Überschätzung von Rewards, die zu einer suboptimalen Policy führen könnte, wird somit unwahrscheinlicher. Zusätzlich werden durch die Verwendung des Minimuns Zustände mit niedrigen Varianzwertschätzungen bevorzugt, wodurch sichere Aktualisierungen der Policy mit stabileren Lernzielen erreicht werden. [21]

Die Netze der Q-Funktionen werden durch die Minimierung des Mean Squared Bellman Error (MSBE), welcher angibt wie gut Q_ϕ die Bellman Gleichung erfüllt, optimiert. Beim SAC ergibt sich somit folgende Loss-Funktion:

$$L(\phi_i,\mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}}\Big[\Big(Q_{\phi_i}(s,a) - y(r,s',d)\Big)^2\Big], \tag{23}$$

wobei der Zielwert definiert ist durch

$$y(r,s',d) = r + \gamma(1-d)\Big(\underset{j=1,2}{min} Q_{\phi'_j}(s',a') - \alpha log\pi_\theta(a'|s')\Big). \tag{24}$$

$\mathcal{D}$ ist der Replay Buffer und d entspricht dem $done$-Signal. Es ist zu beachten, das die Aktionen a' nicht aus dem Replay Buffer, sondern aus der Policy π_θ gesampled werden. [19]

Wie bei anderen Actor-Critic-Verfahren wird anschließend die Policy aktualisiert. Beim SAC wird dazu die *Kullback-Leibler-Divergenz* (KLD) verwendet, welche ein Maß für die Unterschiedlichkeit zweier Wahrscheinlichkeitsverteilungen ist. Die Policy π_θ wird so optimiert, dass die erwartete KLD minimiert wird [18]:

$$J(\pi_\theta) = \underset{s_t\sim\mathcal{D}}{\mathbb{E}}\Big[\underset{a_t\sim\pi_\theta}{\mathbb{E}}\big[\alpha log\pi_\theta(a_t|s_t) - \underset{i=1,2}{min} Q_{\phi_i}(s_t,a_t)\big]\Big] \tag{25}$$

Die Optimierung beinhaltet den Erwartungswert über den Output des Policy-Netzes, wodurch eine normale Backpropagation nicht möglich ist. Daher wird ein Reparametrisierungs-Trick verwendet, welcher den Output des Policy-Netzes mit einem Rausch-Vektor ξ kombiniert, welcher von einer Gaußschen-Verteilung gesampled wird. Normalerweise würde

das Policy-Netz eine Standardabweichung σ und einen Mittelwert μ ausgeben und es würde eine zufällige Aktion gemäß $a \sim \mathcal{N}(\mu, \sigma)$ gesampled werden. Durch den Rausch-Vektor werden die Samples stattdessen durch $a = \mu + \sigma\epsilon$ gebildet, wobei $\epsilon \sim \mathcal{N}(0, 1)$ gilt. Um die Reparametrisierung zu verdeutlichen wird a_t wie folgt definiert: [22]

$$a_t = f_\theta(\epsilon_t; s_t) \tag{26}$$

Gleichung (25) kann nun umgeschrieben werden zu:

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\substack{s_t \sim \mathcal{D} \\ \epsilon_t \sim \mathcal{N}}} \Big[\alpha log \pi_\theta(f_\theta(\epsilon_t; s_t)|s_t) - \mathop{min}_{i=1,2} Q_{\phi_i}(s_t, f_\theta(\epsilon_t; s_t)) \Big], \tag{27}$$

wobei π_θ nun implizit durch f_θ definiert ist [17]. Für weitere Hintergründe und eine tiefgreifendere Erklärung des Reparametrisierungs-Tricks wird auf [23] und [24] verwiesen.

Um den Algorithmus in der Praxis einsetzen zu können, muss die Gaußsche-Verteilung der Aktionen noch auf ein endliches Intervall begrenzt werden. Dazu empfiehlt das originale Paper des SAC die Verwendung des Tangens hyperbolicus (tanh), welche in diesem Kontext auch *Squashing-Funktion* genannt wird [17].

Die bis hierhin erläuterten Grundlagen des SAC reichen nun aus, um den Algorithmus zu implementieren und anzuwenden. Um die Auswahl des Hyperparameters α bei der Entropie zu erleichtern, ist eine zweite Version des SAC ein Jahr später veröffentlicht worden. Anstatt α manuell auszuwählen, schlägt das Paper eine automatische Anpassung vor, indem folgende Gleichung minimiert wird: [18]

$$J(\alpha) = \mathop{\mathbb{E}}_{a \sim \pi_\theta} \Big[-\alpha log \pi_\theta(a|s) - \alpha \bar{\mathcal{H}} \Big], \tag{28}$$

wobei der Parameter $\bar{\mathcal{H}}$ auch Target-Entropie genannt wird und abhängig von der Dimension des Aktionsraumes der Umgebung ist. Das Paper empfiehlt die Verwendung von:

$$\bar{\mathcal{H}} = -dim(\mathcal{A}) \tag{29}$$

Beim BipedalWalker entspricht die Target-Entropie somit dem Wert von -4.

Algorithm 2 Soft Actor-Critic

Initialisiere den Policy Parameter θ
Initialisiere die Q-Funktion Parameter ϕ_1, ϕ_2
Initialisiere ein leeres Replay Buffer D
Kopiere die Q-Funktion Parameter ϕ_1, ϕ_2 als Target Parameter $\phi'_1 \leftarrow \phi_1, \phi'_2 \leftarrow \phi_2$
for episode = 1...M **do**
    Setze die Umgebung zurück
    Der Agent beobachtet den Zustand s
    for steps = 1...T **do**
        Wähle eine Aktion $a \sim \pi_\theta(\cdot|s)$
        Agent führt die Aktion a in der Umgebung aus
        Agent beobachtet den nächsten Zustand s', erhält einen Reward r und ein done-Signal d
        Speichere (s, a, r, s', d) im Replay Buffer D
        if $len(D) \geq batch\_size$ **then**
            Wähle einen zufälligen Mini-Batch $B = (s, a, r, s', d)$ aus D
            Berechne Zielwert für die Q-Funktionen:

$$y(r, s', d) = r + \gamma(1 - d)\Big(\mathop{min}_{j=1,2} Q_{\phi'_j}(s', a') - \alpha log \pi_\theta(a'|s')\Big)$$

            Führe Optimierungsschritt für ϕ_1 und ϕ_2 durch

$$\nabla_{\phi i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s, a) - y(r, s', d))^2$$

            Führe Optimierungsschritt für θ durch

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} (\alpha log \pi_\theta(a|s) - \mathop{min}_{i=1,2} Q_{\phi_i}(s, a))$$

            Aktualisiere den Temperaturparameter α

$$\nabla_\alpha \sum_{(s,a) \in B} (-\alpha log \pi_\theta(a|s) - \alpha \bar{\mathcal{H}})$$

            Update die Target-Netze mit

$$\phi' \leftarrow \rho\phi' + (1 - \rho)\phi,$$

IV. Implementierung

Im Folgendem wird auf die Implementierung und Umsetzung des Deep Q-Learning und Soft Actor-Critic Algorithmus eingegangen.

A. Deep Q-Learning

Das Deep Q-Learning wird analog zu Algorithm 1 umgesetzt. Das Netz des Deep Q-Algorithmus besteht insgesamt aus vier Layern: einer Input-, zwei Hidden- und einer Output-Layer (siehe Abbildung 6). Als Input dient der Zustandsraum des Agenten. Die beiden Hidden-Layer bestehen aus jeweils 64 Neuronen und benutzen als Aktivierungsfunktion die ReLU-Funktion. Die ReLU-Funktion wird definiert durch:

$$ReLU(x) = \begin{cases} x & \text{für } x \geq 0 \\ 0 & \text{für } x < 0 \end{cases} \tag{30}$$

Wenn x also negativ ist, gibt die Funktion 0 zurück, ansonsten den Wert x. Danach folgt die Output-Layer welche der Dimension des Aktionsraumes entspricht. In dieser Schicht wird anstatt einer ReLU-Funktion eine lineare Aktivierungsfunktion eingesetzt.

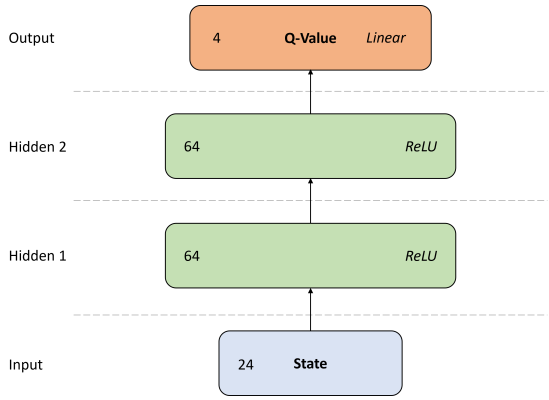


Fig. 6: Netzaufbau Deep Q-Learning

Nachdem die Action-Value-Funktion und die Target Action-Value-Funktion initialisiert wurden, startet der Trainingsprozess. Der Trainingsprozess beinhaltet 5000 Episoden. Zu Beginn jeder neuen Epoche wird die Umgebung zurückgesetzt, sodass der Agent jede Trainingsepoche von vorne startet. Das darauf folgende Preprocessing wird nicht benötigt, da wir mit nicht mit Bildern lernen, sondern mit Zuständen die wir aus der Umgebung erhalten. Innerhalb einer Trainingsepoche kann der Walker maximal 1600 Schritte zurücklegen. Für jeden Schritt wird der Q-Funktion der aktuelle Zustand s des Walkers übergeben und eine Aktion a ermittelt. Die Q-Funktion liefert als Ergebnis einen Vektor mit vier Elementen. Durch die $\epsilon - greedy$ Strategie wählt er mit einer Wahrscheinlichkeit ϵ eine zufällige Aktion aus und mit einer Wahrscheinlichkeit von $1-\epsilon$ die Aktion, die den größten Action-Value-Wert besitzt. Anschließend führt der Walker die ermittelte Aktion a aus und erhält einen Reward durch die Umwelt. Der Reward wird auf den bisherigen Reward der Epoche addiert und die Erfahrung des Walkers wird im Replay Buffer gespeichert. Die Erfahrung besteht aus dem aktuellen Zustand s, der Aktion a, dem Reward r, dem done-Signal d und dem nächsten Zustand s'.

Nachdem Speichern der Erfahrung folgt der eigentliche Lernprozess der Q-Funktion bzw. des neuronalen Netzes. Hierzu wird zunächst ein zufälliger Mini-Batch, mit einer Größe von 32 aus dem Speicher geladen. Daraufhin wird anhand des nächsten Zustands s' der maximale Action-Value-Wert bestimmt. Mit diesem lässt sich der Zielwert y ermitteln, der wiederum als Vektor zum Trainieren des Netzes verwendet wird.

Jedes Gelenk des Walkers kann kontinuierliche Aktionen im Bereich von $A \in [-1, 1]$ ausführen. Da das Q-Learning für diskrete Aktionen vorgesehen ist, haben wir zusätzlich eine Version implementiert, bei dem die Aktionen der einzelnen Gelenke diskretisiert werden. Die Diskretisierung ermöglicht es, den kontinuierlichen Aktionsraum des BipedalWalkers deutlich zu minimieren. Hierbei haben wir uns nach dem Paper [25] für die Einteilung in 7 Diskretisierungsschritte entschieden, da der Unterschied zwischen 7, 11 und 15 Schritten minimal war und es so am wenigsten Aktionen gibt.

B. Soft Actor-Critic

Der SAC ist entsprechend Algorithmus 2 umgesetzt, wobei kleinere Anpassungen erfolgen, um die Umgebung besser lösen zu können. Für die Umsetzung des Algorithmus werden insgesamt fünf neuronale Netze benötigt. Davon entfallen vier für die Q-Funktionen und deren Zielwerte (Critic) und eine für die Policy (Actor).

Die neuronalen Netze für die Q-Funktion bestehen jeweils aus einer Input- und einer Output- sowie aus zwei Hidden-Layer (siehe Abbildung 7). Als Input dient die Kombination aus dem Zustands- und Aktionsraum. Die Hidden-Layer besitzen jeweils 256 Neuronen und nutzen als Aktivierungsfunktion die ReLU-Funktion. Danach folgt die Output-Layer mit einer linearen Aktivierungsfunktion und der Action-Value-Funktion als Ausgabewert. Für die beiden Netze, die die Q-Funktion approximieren, wird der Optimierungsalgorithmus Adam verwendet. Da die Target-Netze keine Optimierungen durchführen, sondern deren Gewichte nur Kopien sind, benötigen sie keinen Optimierungsalgorithmus.

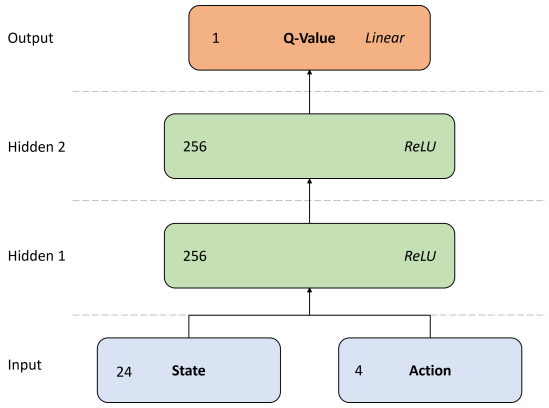


Fig. 7: Critic-Netz

Das neuronale Netz für die Policy ist etwas anders aufgebaut (siehe Abbildung 8). Als Input bekommt es nur den Zustandsraum der Umgebung übergeben. Anschließend folgen drei Hidden-Layer mit jeweils 256 Neuronen und der ReLU-Funktion als Aktivierungsfunktion. Da es sich um eine stochastische Policy handelt besitzt das Netzwerk zwei Outputs, sodass die Verteilung der Aktionen mittels Mittelwert und logarithmischen Standardabweichung abgebildet werden kann. Um zu verhindern, dass die logarithmische Standardabweichung zu groß wird, wird sie auf den Bereich von $[-2, 20]$ begrenzt.

Um eine Aktion aus der Policy zu generieren wird zunächst der beobachtete Zustand s an das Policy-Netz übergeben, um den Mittelwert μ und die logarithmische Standardabweichung $log(\sigma)$ der Wahrscheinlichkeitsverteilung für die Aktionen zu

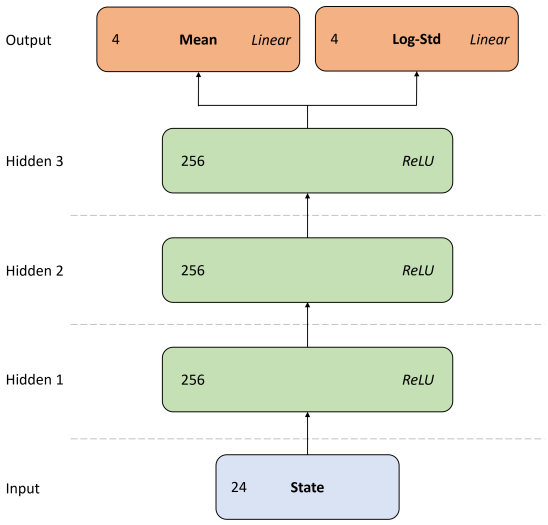


Fig. 8: Actor-Netz

erhalten. Anschließend wird ein Rausch-Vektor ϵ aus einer Normalverteilung gesampled. Die Aktion a kann dann wie folgt berechnet werden:

$$a = tanh(\mu + \sigma\epsilon) \tag{31}$$

Durch den Tangens hyperbolicus sind die Aktionen auf $[-1, 1]$ beschränkt. Da der BipedalWalker den gleichen Wertebereich für seine Aktionen hat, ist keine Skalierung nötig. Andernfalls müsste der Wertebereich noch mit dem maximalen Wert einer Aktion multipliziert werden.

Zudem kann die logarithmische Wahrscheinlichkeit der Aktionen gemäß Anhang C des originalen Papers [17] berechnet werden:

$$log\pi(a|s) = log\lambda(u|s) - \sum_{i=1}^{4} log(1 - tanh^2(u_i)), \tag{32}$$

wobei $u = \mu + \sigma\epsilon$ gilt und λ die zugehörige Dichte-Funktion ist.

Neben den fünf neuronalen Netzen gibt es noch einen Adam-Optimizer, welcher den Temperaturkoeffizienten α gemäß Gleichung (28) anpasst. So kann die Policy in Zuständen, in denen die optimale Aktion unsicher ist, mehr erkunden und in Zuständen, in denen es eine klare Unterscheidung zwischen guten und schlechten Aktionen gibt, deterministischer bleiben.

Um die Exploration weiter zu verbessern wird eine Anpassung eingebaut, die in der originalen Version des SAC nicht vorgesehen ist. Es wird eine Anzahl an Schritten definiert, in denen der Agent zufällige Aktionen durchführt ohne die zu erlernende Policy zu verwenden. Dies hilft den Replay Buffer mit zufälligen Transitionen zu füllen.

Der Lernprozess der neuronalen Netze und des Temperaturkoeffizienten α erfolgt gemäß Algorithmus 2.

V. Auswertung

In dem folgenden Abschnitt werden die Ergebnisse der einzelnen Algorithmen in der BipedalWalker-v3 Umgebung dargestellt. Zusätzlich wird der SAC Algorithmus auf die BipedalWalkerHardcore-v3 Umgebung übertragen und zwei verschiedene Herangehensweisen verglichen.

A. Deep Q-Learning

Die Parameter des Deep Q-Learnings sind in Table III dargestellt. Unser Ansatz lernt mit einer festen Lernrate lr von 0.0005. Der Explorationsfaktor ϵ der $\epsilon - greedy$ Strategie beginnt bei einem Startwert von 1 und reduziert sich bis zum Minimum von ϵ_min 0.05. Jede Episode wird das ϵ mit dem Reduktionsfaktor ϵ_decay multipliziert und verkleinert sich somit kontinuierlich. Die Größe des Replay Buffers wird auf 50000 und die Batch-Größe auf 32 begrenzt. Der Agent lernt solange bis er die 5000 Episoden durchlaufen hat.

TABLE III: Hyperparameter DQN

Parameter	Wert
Lernrate lr	0.0005
Diskontierungsfaktor γ	0.99
Explorationsfaktor ϵ	1
Reduktionsfaktor ϵ_decay	0.999
Min. Explorationswert ϵ_min	0.05
Replay-Buffer-Größe	50000
Batch-Größe	32
Episoden	5000

In Abbildung 9 ist der Reward des Agenten über die Episoden aufgetragen. Der Agent benötigt für 5000 Episoden insgesamt 7 Stunden und 30 Minuten. Der maximale Reward den der Agent erreichen konnte liegt bei -63. Zu Beginn des Lernprozess ist der Reward des Agenten im Durchschnitt deutlich höher als zum Ende hin. Bis zur Episode 1000 liegt der durchschnittliche Reward bei -100 und gegen Ende des Trainings pendelt dieser sich bei einem Wert von -112 ein. Das der Agent zu Beginn besser abschneidet als zum Ende hin, kann auf die $\epsilon - greedy$ Strategie zurückgeführt werden. Am Anfang wählt der Agent durch das hohe ϵ häufig zufällige Aktionen aus, die bessere Ergebnisse liefern als die Aktionen, die der Agent auswählt, um die einzelnen Q-Werte zu maximieren. Der am Ende durchschnittliche stagnierte Reward zeigt, dass der Agent nicht lernt. Auch bei der Verwendung anderer Hyperparameter konnte keine Verbesserung der Ergebnisse erreicht werden.

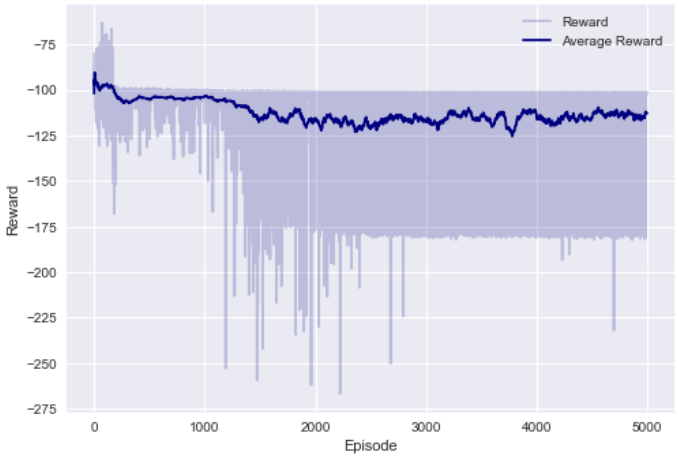


Fig. 9: Reward des DQN-Agenten.

Dies ist auch anhand der ausgeführten Schritte pro Episode ersichtlich (siehe Abbildung 10). Zu Beginn führt der Walker durch das hohe ϵ viele zufällige Aktionen aus und schafft es häufig die maximale Anzahl von 1600 Schritten zu erreichen. Während des Lernprozesses, bei dem ϵ stetig sinkt und somit weniger zufälligen Aktionen ausgeführt werden, sinken auch die durchschnittlich durchgeführten Schritte des Agenten. Die Schritte pendeln sich ungefähr ab Episode 4300 bei einem Wert zwischen 250 und 300 ein. Ab diesem Zeitpunkt fällt der Walker kontinuierlich nach ca. 280 Schritten auf den Boden oder bleibt in einem Spagat hängen, bis die maximale Schrittanzahl erreicht ist.

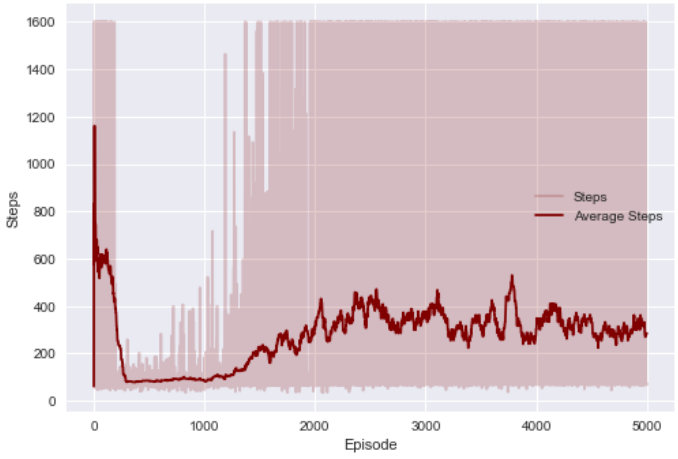
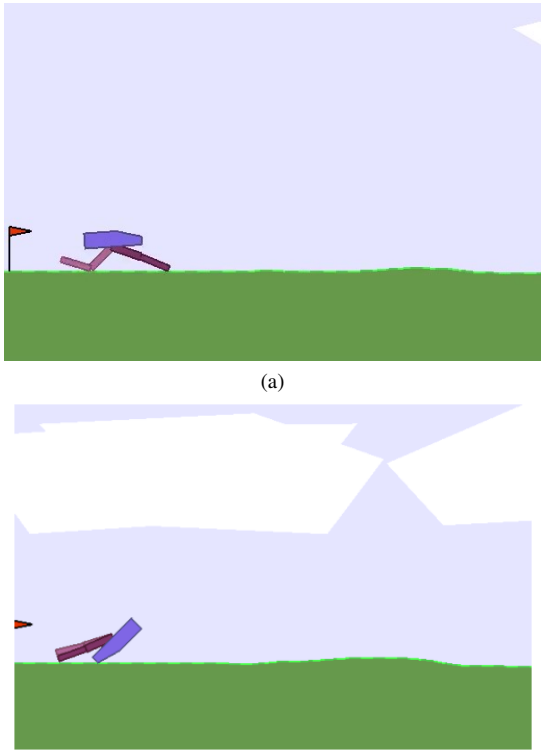


Fig. 10: Steps des DQN-Agenten.

In Abbildung 11 ist die Strategie des Walkers nach dem Training dargestellt. Wie schon anhand des Rewards und der Steps abzulesen, scheitert der Agent beim Laufen lernen. Es resultieren zwei Zustände in denen der Walker endet. Im ersten Zustand bleibt der Walker in einem Spagat hängen und im zweiten Endzustand kippt der Rumpf des Walkers nach vorne, woraufhin der Walker zu Boden fällt. Ein Ansatz zum Optimieren des Deep Q-Learning Algorithmus ist es den kontinuierlichen Aktionsraum des BipedalWalkers zu diskretisieren.



(a)

(b)

Fig. 11: Endzustände des Walkers nach dem Training

Auch bei der Diskretisierung der Aktionen des Walkers konnten keine Verbesserungen festgestellt werden. Der Reward der Diskretisierung ist in Abbildung 12 aufgetragen. Ähnlich wie bei dem vorherigen Ansatz ist der Reward zu Beginn des Trainings am größten, fällt daraufhin ab und stagniert zum Schluss bei einem Wert von -110.

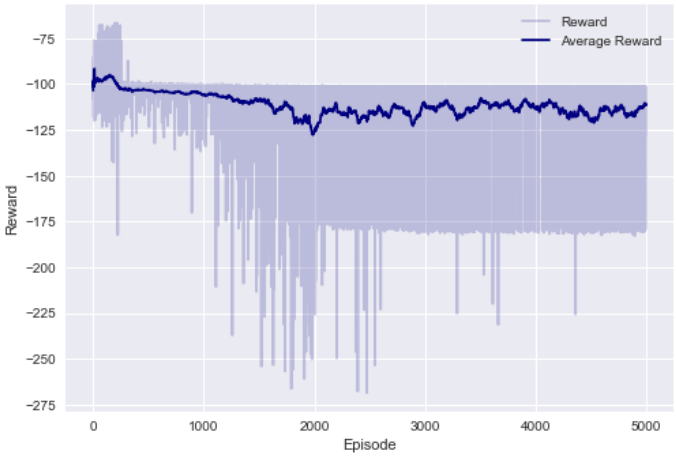


Fig. 12: Reward des DQN Agenten mit Diskretisierung

B. Soft Actor-Critic

Der SAC ist ein Verfahren, welches insbesondere mit der automatischen Anpassung des Temperaturkoeffizienten α sehr stabil bezüglich seiner Hyperparameter ist und somit wenig Feintuning benötigt [18]. Nichtsdestotrotz werden für die Lösung des BipedalWalker einige Hyperparameter aus dem originalen Paper verändert, um deren Einfluss zu beobachten und das Lernverhalten optimieren zu können. In der originalen Version des SAC wird anstatt des Polyak-Koeffizienten ρ der Parameter τ verwendet, welcher dem Wert von $1-\rho$ entspricht. Um unsere Hyperparameter mit denen aus dem originalen Paper besser vergleichbar zu machen, wird nachfolgend auch der Parameter τ für das Polyak Averaging verwendet.

Table IV zeigt die Hyperparameter, die ihren Ursprung aus dem Paper [18] haben und bis auf die Lernrate und den Polyak-Koeffizienten nicht verändert werden. Zusätzlich befinden sich in Table V verschiedene Kombinationen der Lernrate lr und des Polyak-Koeffizienten τ, die nachfolgend ausgewertet werden.

TABLE IV: Hyperparameter SAC [18]

Parameter	Wert
Lernrate lr	0.0003
Diskontierungsfaktor γ	0.99
Polyak-Koeffizient τ	0.005
Target-Entropie $\bar{\mathcal{H}}$	-4
Replay-Buffer-Größe	1000000
Batch-Größe	256
Neuronen Hidden-Layer	256
Aktivierungsfunktion	ReLU
Optimizer	Adam

TABLE V: Ausgewählte Kombinationen der Hyperparameter

Kombination	Lernrate lr	Polyak-Koeffizient τ
K1	0.0003	0.005
K2	0.0003	0.01
K3	0.0001	0.01

Schon bei der Verwendung der originalen Hyperparameter aus dem Paper (K1) in unserer Implementierung des SAC wird der BipedalWalker erfolgreich gelöst. Wie in Abschnitt II-A erwähnt, gilt die Umgebung als gelöst, wenn der BipedalWalker einen durchschnittlichen Reward von mindestens 300 über 100 aufeinander folgenden Episoden erreicht. Mit den gewählten Hyperparametern erfolgt dies nach 718 Episoden (siehe Abbildung 13).

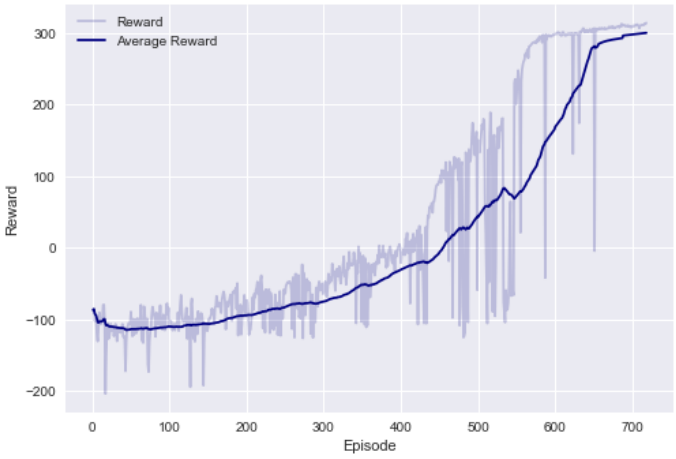
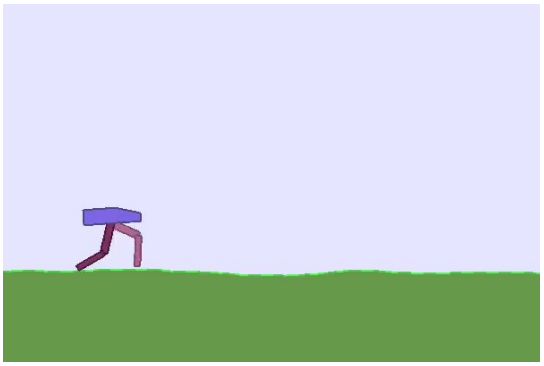


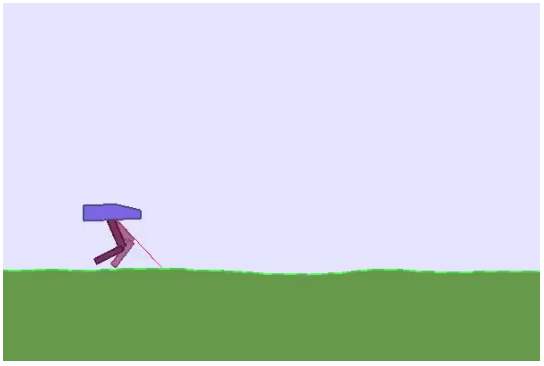
Fig. 13: Rewards des SAC-Agenten K1

Betrachtet man die durchgeführten Schritte pro Episode wird auch das Lernverhalten des BipedalWalker ersichtlich (siehe Abbildung 14). Zu Beginn fällt der Walker fast ausschließlich hin, sodass weniger als 200 Schritte durchgeführt werden. Anschließend verbleibt der Walker immer öfter im Spagat, wodurch die maximale Anzahl an Schritten pro Episode erreicht wird. Vom Spagat aus lernt er seine Beine nach vorne zu ziehen, bis er irgendwann aufrecht laufen kann (siehe Abbildung 15. Dabei fällt der Walker immer seltener um und schafft es schließlich bis zum Ende zu laufen, wodurch Rewards von über 300 erreicht werden. Die benötigten Schritte zum Erreichen des Ziels sinken dabei stetig.

Fig. 14: Schritte pro Episode des SAC-Agenten K1



(a)



(b)

Fig. 15: Startposition (a) und Endposition (b) eines Schrittes des Walkers nach erfolgreichem Training

Lässt man den Walker weitere 500 Episoden trainieren, so optimiert er sein Laufverhalten weiter, wodurch die benötigten Schritte sinken und der Reward auf teilweise bis zu 325 ansteigt (siehe Abbildung 16). Da es bei den Ranglisten von OpenAI aber nicht um den maximalen durchschnittlichen Reward geht, sondern um die kürzeste Anzahl von Episoden bis der durchschnittliche Reward von 300 erreicht ist, wird der Lernprozess nachfolgend beim Erreichen eines durchschnittlichen Rewards von 300 abgebrochen.

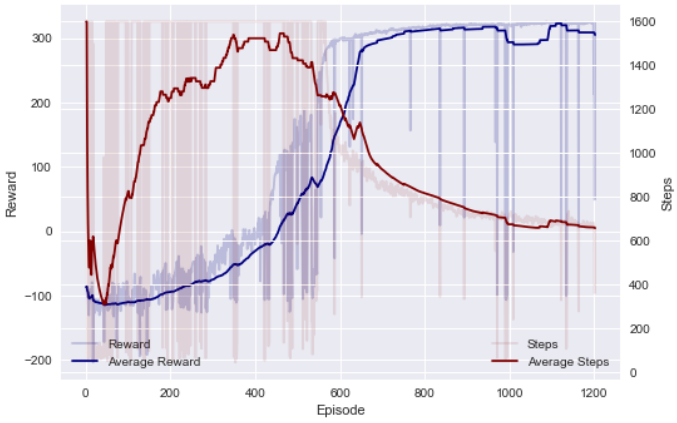


Fig. 16: Schritte und Rewards des SAC-Agenten K1 über 1202 Episoden

Die originalen Parameter (K1) reichen bei unserer Implementierung schon aus, um die aktuelle Rangliste der BipedalWalker-v3 Umgebung anzuführen [26]. Allerdings ist eine Verbesserung durch eine Veränderung des Polyak-Koeffizienten τ auf 0.01 (K2) möglich, sodass der durchschnittliche Reward von 300 über 100 Episoden schon nach Episode 474 erreicht wird (siehe Abbildung 17).

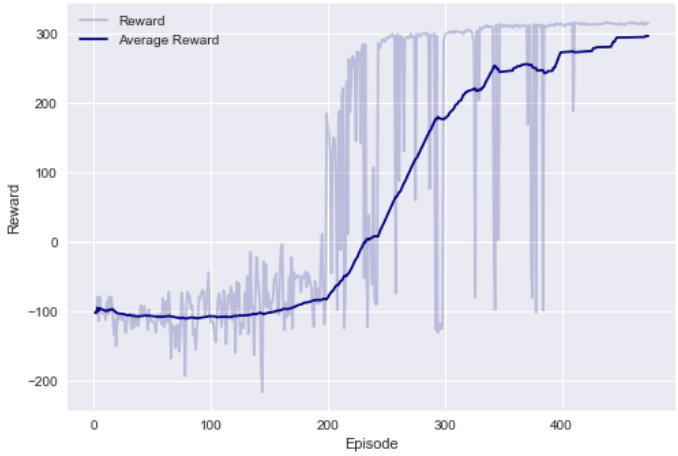


Fig. 17: Rewards des SAC-Agenten K2

Eine Veränderung der Lernrate auf 0.0001 (K3) konnten das Ergebnis vom Agenten K2 nicht übertreffen (siehe Abbildung 18). Auch eine Erhöhung der Lernrate auf 0.0005 oder des Polyak-Koeffizienten auf 0.02 führten nach über 800 Episoden nicht zum Ziel und wurden daher abgebrochen.

Fig. 18: Rewards der SAC-Agenten K3

Das Lösen des BipedalWalkerHardcore ist wie in Abschnitt II-B beschrieben, deutlich anspruchsvoller als die Standard-Version. So ist auch von einer längeren Trainingszeit auszugehen. Beim BipedalWalker hatte es ausgereicht für die gesamte Umgebung ein Laufverhalten zu lernen um ans Ziel zu kommen. Da es beim BipedalWalkerHardcore auch Hindernisse in

Form von Treppenstufen, Gruben und Stolperfallen gibt, muss der Walker neben dem Laufen noch verschiedene Techniken lernen, um diese Hindernisse zu überwinden. Daher ist es besonders wichtig, eine gute Explorations-Strategie zu verwenden, um für jedes Hindernis die geeignete Technik lernen zu können.

Für die Auswertung der BipedalWalkerHardcore Umgebung werden zwei unterschiedliche Herangehensweisen untersucht und miteinander verglichen. Zunächst wird ein Agent (A1) betrachtet, der die Umgebung von Grund auf neu trainiert. Anschließend wird versucht das Training zu beschleunigen, indem ein Agent (A2) verwendet wird, der schon auf dem BipedalWalker trainiert wurde. Die Grundlage für den Agenten stellt dabei der Agent K2 aus der BipedalWalker Umgebung dar, der einen durchschnittlichen Reward von 300 in 474 Episoden erreichen konnte. Für den Agenten A1 werden die Neuronen der Hidden Layer auf 400 erhöht und eine Lernrate von 0.0005 ausgewählt. Die für das Training der beiden Varianten verwendeten Hyperparameter sind in Table VI aufgeführt. Die in Abschnitt IV-B erwähnten zufälligen Schritte zu Beginn des Trainings werden beim bereits trainierten Agenten weggelassen, da die Unterschiede der beiden Umgebungen so ohnehin nicht erkundet werden können.

TABLE VI: Hyperparameter SAC BipedalWalkerHarcore

Parameter	A1	A2
Lernrate lr	0.0005	0.0003
Diskontierungsfaktor γ	0.99	0.99
Polyak-Koeffizient τ	0.01	0.01
Target-Entropie $\bar{\mathcal{H}}$	-4	-4
Replay-Buffer-Größe	1000000	1000000
Batch-Größe	256	256
Neuronen Hidden-Layer	400	256
Aktivierungsfunktion	ReLU	ReLU
Optimizer	Adam	Adam

Abbildung 19 zeigt die Rewards des Agenten, der zum ersten mal trainiert wird, über einen Zeitraum von 1945 Episoden. Das Training dauerte für diese Episoden bereits über 100 Stunden, weshalb es bis zur Abgabe dieser Arbeit nicht zu Ende laufen konnte. Nichtsdestotrotz lässt sich das Lernverhalten anhand der durchgeführten Episoden analysieren. Die ersten 200 Episoden ähneln sehr dem Verlauf des normalen BipedalWalkers mit gleichen Hyperparametern (siehe Abbildung 17). Dies liegt daran, dass der Walker auch in der Hardcore-Version erst lernt einen Spagat zu machen und aus dieser Stellung heraus, das Laufen erlernt. Beim BipedalWalkerHardcore ist danach allerdings ein Abfall der Reward-Kurve zu erkennen, was auf die hinzugekommenen Hindernisse zurückzuführen ist. Anschließend steigt der erreichte Reward pro Episode kontinuierlich an. Dabei ist eine deutlich höhere Streuung als bei der Standard-Version zu erkennen, da der Walker trotz eines guten Laufverhaltens sehr häufig über ein Hindernis stolpert. Der höchste durchschnittliche Reward wurde nach ca. 1480 Episoden erreicht, danach ist zeitweise ein kleiner Abstieg zu erkennen. Da der Agent es regelmäßig schafft einen Reward von über 300 zu erreichen, ist davon auszugehen, dass er mit genügend Trainingszeit auch einen durchschnittlichen Reward von 300 erreichen wird.

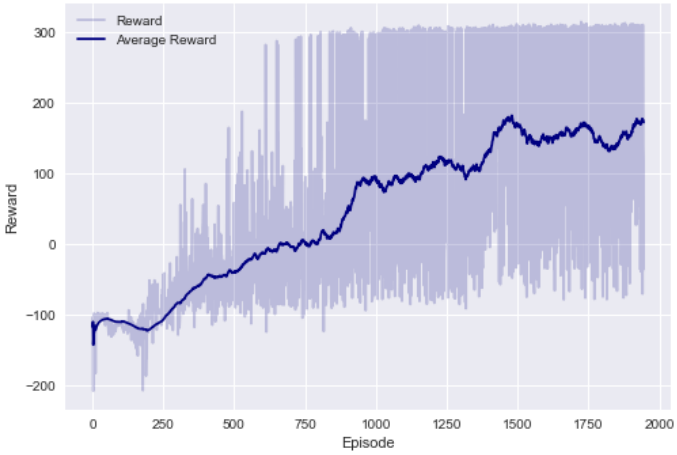
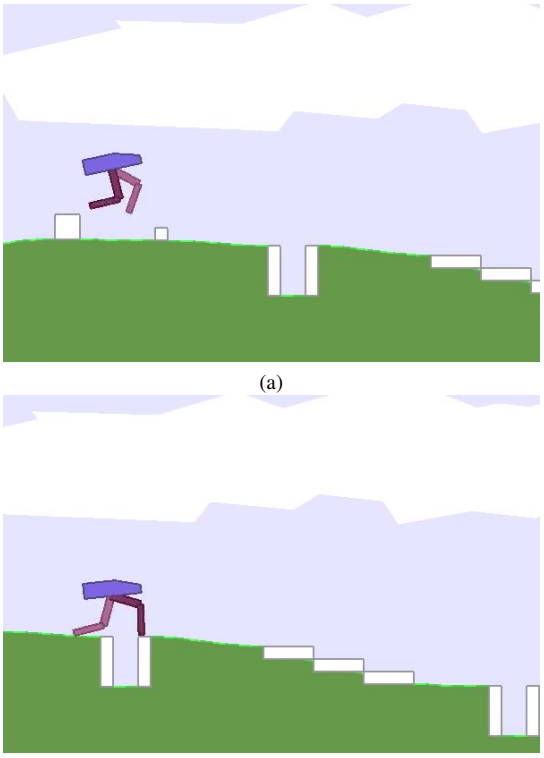


Fig. 19: Rewards des Agenten A1, der von Grund auf die BipedalWalkerHardcore-Umgebung trainiert

Betrachtet man den trainierenden Agenten nach den 1945 Episoden lässt sich auch erkennen, in welchen Situationen er noch trainiert werden muss. So schafft es der Agent fast jede Fallgrube und Treppe zu überwinden, scheitert jedoch teilweise beim Überwinden von Stolperfallen, insbesondere wenn sich zwei Stolperfallen nah beieinander befinden. Abbildung 20 zeigt den Agenten beim erfolgreichen Überqueren von zwei Hindernissen. Auffällig ist auch, dass sich das Laufverhalten grundsätzlich vom normalen BipedalWalker unterscheidet. Während der Walker in der Standard-Version eine Technik lernt, bei dem ein Bein immer vorne ist und eine Vorwärtsbewegung durch viele kleine Schritte entsteht, verfolgt der Agent in der BipedalWalkerHardcore eine deutlich komplexere Technik. Dabei führt der Walker ein menschenähnliches Laufverhalten aus, bei dem die Beine bei jedem Schritt gewechselt werden. Zudem kommt es bei verschiedenen Situationen zu einer Veränderung der Lauftechnik vor, um die jeweiligen Hindernisse zu überwinden.

Wie zu erwarten, unterscheidet sich das Lernverhalten des Agenten, der bereits auf den BipedalWalker trainiert wurde, deutlich von einem neuen Agenten (siehe Abbildung 21). Allerdings ist das Ergebnis anders, als man zunächst erwarten würde. Denn obwohl der Agent schon zu Beginn des Trainings laufen kann, steigt der durchschnittliche Reward sichtbar langsamer als bei einem neuen Agenten. Eine Erklärung für dieses Phänomen könnten die bereits erwähnten Unterschiede der Lauftechniken zwischen der Standard- und der Hardcore-Version sein. So benötigt der Agent länger seine Technik auf die Hindernisse anzupassen, als eine neue Technik von Grund auf zu lernen. Eine andere Erklärung könnte auch die höhere Lernrate oder die größeren Hidden-Layer sein. Um dies genauer zu untersuchen, sollten mehr Versuche mit unterschiedlichen Hyperparametern durchgeführt werden. Eventuell sollte auch die automatische Anpassung des Tem-

peraturkoeffizienten angepasst werden, wenn eine Umgebung nicht von Grund auf trainiert wird. Nichtsdestotrotz schafft es auch die zweite Herangehensweise einen Reward von über 293 zu erreichen und somit die Umgebung inklusive der zufälligen Hindernisse zumindest in einzelnen Episoden zu überqueren.



(a)

(b)

Fig. 20: Der Agent beim Überwinden einer Stolperfalle (a) und einer Fallgrube (b)

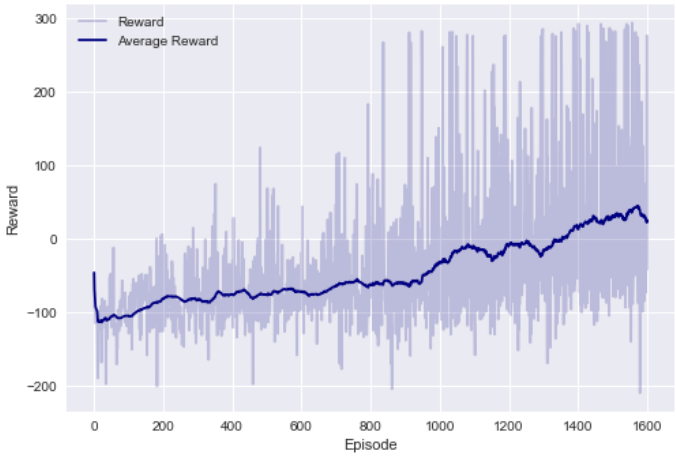


Fig. 21: Rewards des Agenten A2, der bereits auf dem BipedalWalker trainiert wurde

VI. Fazit und Ausblick

Das Ziel dieser Arbeit war es, sowohl einen Deep Q-Learning- als auch einen Soft Actor-Critic-Algorithmus zu implementieren und die BipedalWalker-v3 Umgebung von OpenAI zu lösen. Aufgrund des kontinuierlichen Aktionraumes der Umgebung konnte mithilfe des Deep Q-Learning-Ansatzes kein Lernverhalten des Agenten erreicht werden. Auch durch eine Diskretisierung der Aktionen konnte keine Verbesserung festgestellt werden. Eine Erweiterung des Deep Q-Learnings durch Methoden wie das Double Q-Learning erscheint aufgrund der schlechten Ergebnisse nicht zielführend.

Daher wurde der Fokus auf den Soft Actor-Critic gelegt, welcher auf dem Gebiet des bestärkenden Lernens für kontinuierliche Aktionen zu den modernsten Methoden gehört. Es wurde eine Adaption der neuesten Version des SAC implementiert, welche eine automatische Anpassung des Temperaturkoeffizienten besitzt. Durch unsere SAC-Implementierung mit den Hyperparametern aus dem originalen Paper des Algorithmus konnte die BipedalWalker-v3 Umgebung in 718 Episoden gelöst werden. Es wurden zusätzlich noch verschiedene Variationen der Hyperparameter getestet, wodurch sich das Ergebnis auf 474 Episoden verbessern konnte und uns auf Platz 1 der OpenAI-Rangliste bringt.

Um die Grenzen des SAC zu erkunden, wurde der Algorithmus ebenfalls in der BipedalWalkerHardcore-v3 Umgebung getestet, welche das Erreichen des Ziels durch zufällige Hindernisse weiter erschwert. Dabei wurden zwei verschiedene Agenten verglichen: ein Agent, der bereits auf der BipedalWalker Umgebung trainiert wurde und ein Agent, welcher neu trainiert wird. Aufgrund der hohen Rechenzeit konnten die Experimente nicht vollständig durchgeführt werden. Es konnte allerdings gezeigt werden, dass bei unseren Parametern ein Agent, der von Grund auf neu trainiert wird in weniger Episoden einen höheren Reward erhält. Durch eine andere Wahl der Hyperparameter oder eine Veränderung der Explorations-Strategie, könnte das Ergebnis eventuell auch anders ausfallen. Dies gilt es weiter zu untersuchen.

Um den Agenten noch schneller und zuverlässiger auf die Umgebung trainieren zu lassen, sollten in einem möglichen Folgeprojekt weitere Variationen der Hyperparameter untersucht werden. So könnten neben der Lernrate und dem Polyak-Koeffizienten auch das Replay Buffer oder der Aufbau der neuronalen Netze eventuell optimiert werden. Auch Anpassungen des SAC durch andere Policy Optimization Ansätze, könnten das Lernverhalten weiter verbessern [27]. Zuletzt wäre es sinnvoll die Trainingsversuche des BipedalWalkerHardcore, die aus Zeitgründen unterbrochen wurden, fortzuführen.

References

[1] OpenAI. *BipedalWalker v2*. URL: https://github.com/openai/gym/wiki/BipedalWalker-v2 (visited on 03/11/2022).

[2] OpenAI. *bipedal_walker.py*. URL: https://github.com/openai/gym/blob/master/gym/envs/box2d/bipedal_walker.py (visited on 03/11/2022).

[3] Jörg Frochte. *Maschinelles Lernen: Grundlagen und Algorithmen in Python*. Carl Hanser Verlag GmbH Co KG, 2019.

[4] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3 (1992), pp. 279–292.

[5] Long-Ji Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine learning* 8.3 (1992), pp. 293–321.

[6] Hado Hasselt. "Double Q-learning". In: *Advances in neural information processing systems* 23 (2010).

[7] Hado Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *arXiv:1509.06461v3* 23 (2015).

[8] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[9] OpenAI. *Proof for Using Q-Function in Policy Gradient Formula*. URL: https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof2.html (visited on 03/11/2022).

[10] Maxim Lapan. *Deep Reinforcement Learning. Das umfassende Praxis-Handbuch: Moderne Algorithmen für Chatbots, Robotik, diskrete Optimierung und Web-Automatisierung inkl. Multiagenten-Methoden*. MITP-Verlags GmbH & Co. KG, 2020.

[11] OpenAI. *Intro to Policy Optimization*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html (visited on 03/11/2022).

[12] Vijay Konda and John Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems* 12 (1999).

[13] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[14] Hao Dong et al. *Deep Reinforcement Learning*. Springer, 2020.

[15] Vıctor Uc-Cetina. "A Novel Reinforcement Learning Architecture for Continuous State and Action Spaces." In: *Advances in Artificial Intelligence (16877470)* (2013).

[16] Csaba Szepesvári. "Algorithms for reinforcement learning". In: *Synthesis lectures on artificial intelligence and machine learning* 4.1 (2010), pp. 1–103.

[17] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[18] Tuomas Haarnoja et al. "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905* (2018).

[19] OpenAI. *Soft Actor-Critic*. URL: https://spinningup.openai.com/en/latest/algorithms/sac.html (visited on 03/11/2022).

[20] OpenAI. *Deep Deterministic Policy Gradient*. URL: https://spinningup.openai.com/en/latest/algorithms/ddpg.html (visited on 03/11/2022).

[21] Scott Fujimoto, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596.

[22] Petros Christodoulou. "Soft actor-critic for discrete action settings". In: *arXiv preprint arXiv:1910.07207* (2019).

[23] Ava Soleimany. *Deep Generative Modeling, Lecture 4*. Massachusetts Institute of Technology. Feb. 2020. URL: https://www.youtube.com/watch?v=rZufA635dq4&ab_channel=AlexanderAmini (visited on 03/11/2022).

[24] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3 (1992), pp. 229–256.

[25] Yunhao Tang and Shipra Agrawal. "Discretizing continuous action space for on-policy optimization". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 5981–5988.

[26] OpenAI. *Leaderboard BipedalWalker-v3*. URL: https://github.com/openai/gym/wiki/Leaderboard#bipedalwalker-v3 (visited on 03/11/2022).

[27] Zhenyang Shi and Surya PN Singh. "Soft Actor-Critic with Cross-Entropy Policy Optimization". In: *arXiv preprint arXiv:2112.11115* (2021).